CSC207 Design Document (Phase 1)

Ritik Kothari, Parth Amin, Petar Jovasevic, Azka Azmi, Meet Patel, Aryaman Modi

15 November 2021

Table of Contents

1 Updated Specification

1.1 Project Domain

Implement a fully functioning chess game (with additional features such as a leaderboard) with the correct logic and game rules. This game will be presented to the user using a clean and organized command-line GUI to facilitate an efficient user experience.

In Phase 0, our team created the chess board and also the 32 total pieces that are initialized. We followed the scenario walkthrough and programmed the first move, which covered a pawn moving either one or two spaces on the board.

In Phase 1, we expanded this to cover the entire course of a typical chess game, with only castling and a leaderboard system left to implement. Leaderboard features would include a system that keeps track of player moves until checkmate is achieved, with lower total player moves implying a higher position on the leaderboard. We will also consider a time tracker, in which the amount of time taken to achieve checkmate will be recorded, with high scores being dealt with in a similar fashion to the player moves.

Further steps for Phase 2 would be to implement the aforementioned features, as well as consider implementing advanced chess moves such as en passant. The code will be polished up, with care being taken to ensure proper Javadoc and other code etiquette.

1.2 Project Domain Rules

The rules of our project domain remain unchanged from Phase 0. To reiterate, we list them below:

Chess is a two-player strategy game that takes place over an 8 x 8 board with one player controlling 16 black pieces and the other controlling 16 white pieces. Beginning with the player controlling the white pieces, each player makes one move per turn with the goal of capturing the opponent's King. The rules for each piece are as follows:

- Pawns: can move forward once (but may move forward twice on their first move of the game) and attack diagonally.
- Kings: can move and attack one block in any direction.
- Queens: can travel any number of blocks in any direction, attacking the block on which they land on.
- Bishops: can only move any number of blocks diagonally, attacking the block on which they land on.
- Rook: can move any number of blocks vertically and horizontally, attacking the block on which they land on.
- Knights: can travel two blocks horizontally and one block vertically (or vice versa), attacking the block on which they land.

White pieces are represented by lowercase names while black pieces are represented by uppercase names. Kings are represented using unit code. If a player's king is in a position to be attacked, then they are in check. If there are no possible moves the player in check can make to break the check, then they lose and therefore, the other player wins.

For Phase 2, we will consider implementing the en passant move, which involves the pawn piece. En passant is a special pawn capture that can only happen right after a pawn moves two squares from its starting position, in the instance where it could have been captured by an enemy pawn if it had advanced only one square. The opponent captures the just-moved pawn "as it passes" through the first square. Regardless of whether the pawn had advanced only one square or the enemy pawn had captured it normally, the result is the same. The en passant capture must be made on the very next turn or the capture is no longer valid.

2 Class Diagram

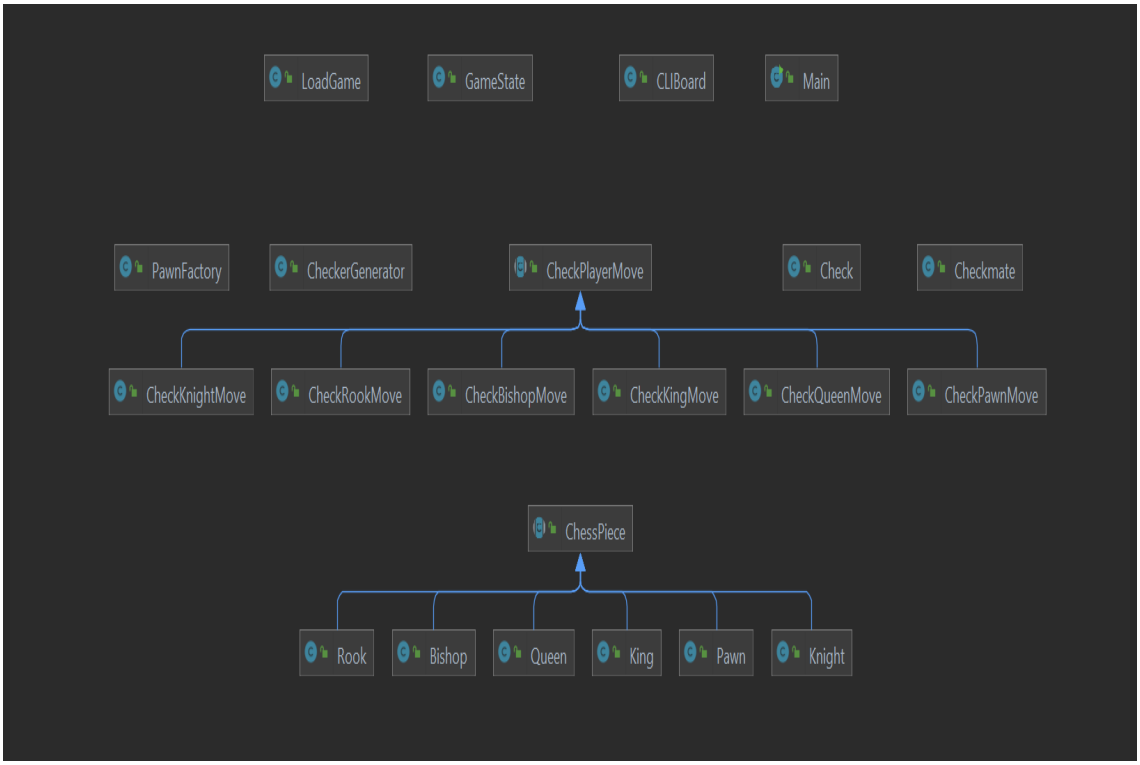

Figure 1: UML Class Diagram for Chess Game

Figure 1 shows the UML Class Diagram

3 Design Decisions

3.1 Using Factory Methods

The first design decision we made was to consider using factory methods. We decided to use the same generic factory method to generate everything instead of using personalized ones, which saved time when it came to the implementation of other features.

We settled on two distinct methods, one for pawn transformation and the other for checker generation.

3.2 Implementing the UI

Another major design decision we made was deciding how to implement the UI. We decided to leave the UI as is from Phase 0, which consisted of the command-line UI with letters representing the various pieces.

Our reasoning for this was that there were additional features that took precedence over the UI, and so we decided to devote more resources towards the implementation of those. The command-line UI is already an efficient method of displaying the state of the board, so we felt that there was no need to modify it.

3.3 Implementing Check and Checkmate

The final major design decision we made was figuring out how to implement the check and checkmate conditions.

During Phase 1, we came up with various ideas on how to implement Check. We considered the case where we determine if the King's location falls for any valid moves of an opposing piece, and also the case where we determine if there are opposing pieces in the King's path that can capture it.

We decided to implement the case where we determine if there are opposing pieces in the King's path that can capture it. This method was chosen because we saw it as more efficient compared to the other one, in that checkers must only be implemented for the King class rather than various pieces.

4 Adherence to Clean Architecture

4.1 Adherence

Our project is divided into different layers while adhering to Clean Architecture. We have the Entities layer, which are the various chess pieces, and as the inner core of our domain, they have no dependencies.

Next, we have the Use Cases, which is the application layer that manipulates the Entities. The Use Cases are responsible for determining the list of possible valid moves for the various chess pieces.

We then have the Controller layer, which interacts with our implementations in the previous layer, the Use Cases. Examples of classes that exist in the Controller layer are GameState and LoadGame.

Finally, we have the CLIBoard class which is part of the fourth layer of our project, the Interface layer. The CLIBoard outputs the current state of the board and displays the board after each valid player move.

4.2 Violations

Currently, all Use Cases (with the exception of CheckerGenerator) import the Controller GameState, which violates Clean Architecture.

A possible solution for this violation could be to create an abstract class or interface that abstracts the required properties of GameState, declare that GameState implements/extends this, and have the Use Cases interact with this new class/interface instead.

Another violation is within the Checkmate class, which is a Controller that interacts predominantly with GameState. The issue here is that Checkmate directly interacts with ChessPiece objects and it's properties within it's methods. This is in violation of SOLID principles as a Controller should not be directly manipulating entities, and could have been easily fixed if our implementation added methods within GameState that could deal with these entities instead of Checkmate, that Checkmate could then call upon.

5 Consistency with SOLID Design Principles

5.1 Consistency

We have implemented our project while adhering to the SOLID design principles. The different layers in which we have divided our project goes on to show the Single Responsibility Principle. Each class in our project performs a unique function, i.e. it only has one responsibility at hand. This has assisted in unit testing our code as we have generated unique tests to check each individual functionality of our project.

Our project adheres to the Open-closed principle of the SOLID design principles, as we will be extending our code to account for advanced features such as the "En Passant" move and "Castling". However we would do so by utilizing the functions we already have, thus we would be extending the scope of our program by building on it, instead of modifying what we already have.

5.2 Violations

As stated above, the Use cases all import the Controller GameState, which is not consistent with the Dependency Inversion Principle of the SOLID design principles.

Possible solutions for this will be explored during Phase 2.

6 Packaging Strategies

Our project uses the packaging by layers strategy. We have separated our project into different layers in adherence to Clean Architecture.

The individual chess pieces: Bishop, King, Knight, Pawn, Queen and Rook, and their abstract parent class Chess Piece are all stored in the Entities Folder. The various move checkers for the different chess pieces are stored in the Use Cases Folder. The Controller: GameState and LoadGame is stored in the Others folder along with the CLIBoard and Main.

7 Design Patterns

7.1 Simple Factory

The Pawn Factory in the Use cases utilizes the Simple Factory Pattern.

The Simple Factory Pattern describes a class that has one creation method with a large conditional that chooses which product class to instantiate and return based on method parameters.

The class Pawn Factory uses the Simple factory Pattern to return an instance of Chess Piece. It takes in a pawn, and the string literal representing the chess piece it is to be transformed into as parameters. It then returns the Chess Piece the pawn was to be transformed into with the attributes of the pawn. For example, if the pawn is to be transformed into a rook, our method would return a rook with the attributes of the pawn.

7.2 Memento

We considered implementing the Memento design pattern in the form of an undo/redo option for chess moves during Phase 1, but we decided to hold off on a possible implementation of this until Phase 2.

An undo/redo feature would take into account one of the major intents of Memento, which is to avoid the violation of encapsulation, so that capturing and externalizing an object's internal state allows it to be returned to this state later. In the instance of programming our chess game, the Controller layers would be modified to save a state of the game prior to a move being made. If a player then decides to withdraw their move, the previous state remains accessible to bring up as the current state of the game.

8 Review of Progress

8.1 Summary of Updated Project Specification

The updated project specification describes the project domain, with progress during each phase outlined (see Section 1.1). It then outlined the rules of chess, which remained unchanged throughout Phase 0 and Phase 1 (see Section 1.2).

Expanding upon the project scenario walk-through from Phase 0, the project's progress has reached a point where a typical chess game can be played without exercising any advanced moves (i.e castling, en passant).

As mentioned above, the program implements the features of a typical chess game. During Phase 2, work will be done to implement the castling move and work involving the implementation of the en passant move will be considered. A leaderboard system will also be implemented.

8.2 Summary of Successful Design Features

To reiterate, some successful design features from Phase 0 include:

- Creating an abstract class called ChessPiece to model the behaviour and common functionalities of a chess piece
- Breaking down a large class into smaller classes to avoid over-complication and excessive responsibilities or functionality (adheres to the SOLID principles)
- Creating a string value that represents our chess game, and all 32 chess pieces. This was then printed out successfully into the command line as a UI

During Phase 1, additional features implemented include:

- Methods that cover the movement of every chess piece
- Methods to capture chess pieces
- Check and Checkmate states for the game
- The Simple Factory design pattern was followed for the creation of the Pawn Factory class

Next steps for Phase 2 detail the addition of advanced chess moves and a leaderboard system (see Section 8.1).

8.3 Open Questions

Some open questions our group struggled with during Phase 0 include:

1. Choosing the most consistent implementation to handle the way pieces are moved such that it abides by the principles of Clean Architecture.
2. Had trouble deciding on the design of handling check and checkmate conditions (Ensuring no possible moves can be made to declare checkmate, preventing invalid moves when in check, etc.).
3. Had trouble deciding on potential extensions of the application like ELO score, history tracking and how that data can be efficiently stored and manipulated.
4. Determining the most efficient way to verify whether a given move is valid.

The above questions were largely answered during Phase 1, with our solutions outlined below:

1. We decided to implement Use Cases that determine the validity of moves and had our Controller class–GameState–make the necessary modifications.
2. For the design of handling check and checkmate, we decided on checking to see if there are pieces in the King's path that can capture it (see Section 3.3 for details).
3. We settled on the implementation of a leaderboard of least moves and least time taken to reach checkmate for possible extensions of the application. Specifics regarding this decision will be addressed during Phase 2.
4. To determine ways to verify the validity of a given move, we implemented several checkers that check a move's validity according to the rules outlined in Section 1.2.

As would be expected, the solutions of pending questions from Phase 0 invited the formation of several new questions that we hope will be answered in Phase 2. Some of those questions are listed below:

- Of the violations mentioned for the SOLID principles and Clean Architecture, which can be remedied without greatly altering the code and which cannot? For those that cannot, what parts of the code must be changed to avoid the violation?
- The rules surrounding the en passant move apply to an extremely specific situation. Implementation of en passant in our program would require the creation of multiple additional checkers for several conditions. For how obscure of a move en passant is, is it worth devoting resources towards its development that can otherwise be used for other aspects of the program?

- Concerning the addition of the leaderboard system, there are multiple factors that must be considered.
  - Our intent is to create a leaderboard system that carries over through many games so that the user can notice any trends in their play. How exactly would this be implemented?
  - We have also considered the deletion and addition of several users so that the leaderboard can keep track of multiple players. What would an outline of the addition and deletion feature look like?
  - Finally, we would require methods that keep track of the number of moves made and time taken to reach checkmate. What would these methods look like, and how would information from them be extracted so that they can be displayed in the leaderboard?

8.4 Summary of Group Member Roles and Next Steps

Ritik - Worked on the design document and suggested ideas for the implementation of several Phase 1 features. Will work on implementing the leaderboard system in Phase 2.

Parth - Worked on the gameState to allow for other pieces to be moved. Worked on reading and writing the current state to a text file to store and also contributed to testing and made test cases for different chess pieces to ensure the that the game works properly for different pieces. Will continue to work on these features and create more advanced test cases for more cumbersome edge cases.

Meet - Worked on implementing unit tests pertaining to the validity of the movements of each Chess piece, as well as testing if the program can accurately identify when the King piece is in check. Will work on adding more advanced test cases for features to later be implemented in Phase 2 such as "castling" or "en passant."

Petar - Implemented the Use Cases that pertain to the movement of pieces across the board: CheckPlayerMove and its subclasses. Refined Entities. Implemented the CheckerGenerator class, which is used in GameState to instantiate the appropriate Use Case when checking the validity of a chess move.

Azka - Worked on implementing Check, Checkmate and Factory methods for our implementation, as well as tests for these classes. Will work on rewriting/refactoring the Check class to be more efficient and implementing more advanced chess moves.

Aryaman - Worked on the design document and contributed to the development of the various move checker classes. Will work on methods that keep track of the number of moves made and time taken for the completion of a game in Phase 2.